H.E.B. Ecole Superieur d'Informatique

Laboratoire de C++ : projet 2

Starlight

Auteurs :
Paul Kriwin
Simon Placentino

Titulaire du cours :
Dr. Romain Absil

21 avril 2015

Table des matières

Chapitre 1

Introduction

Chapitre 2

Quelques conventions utilisées

2.1 C++

Ce projet a été bati à l'aide de **Qt Creator 5.4** [1], pour l'accès à la bibliothèque graphique, et compilé à l'aide de **g++** [2] dans sa version 4.8 (et compatible 4.9). La norme ISO/IEC 14882 :2011, aussi appellée c++11, est celle utilisée. A cela s'ajoute certains "flags" de compilation

g++ -std=c++11 -Wextra -Wall -pedantic-errors

- -std=c++11 pour travailler en c++11,
- -Wextra qui nous permet d'avoir des messages d'avertissements,
- -Wall qui nous permet d'avoirtous les messages d'avertissements,
- -pedantic-errors qui transforme certains "warnings" en erreurs.

2.2 Contenu des fichiers

Chaque fichier contiendra les en-têtes, ou le code source, d'une seule et unique classe ou d'un seul et unique namespace.

2.3 Nom des fichiers

Le nom des fichiers est entièrement écrit en minuscule et porte le nom de la classe, ou du namespace qu'il contient.

2.4 Nom des fichiers de test

Le nom des fichiers de test est composé du nom de la classe ou du namespace testé suivi de "test"

1. `http://www.qt.io/developers/`
2. `https://gcc.gnu.org/`

```
nuketest.cpp
mirrortest.cpp
```

2.4.1 headers

Les headers porteront l'extension **hpp**.

2.4.2 sources

Les fichiers sources porteront l'extension **cpp**.

2.5 Nom des namespace

Le nom d'un namespace sera exclusivement en minuscule.

2.6 Nom des classes

Les classes commencent toutes par une majuscule pour continuer, ensuite, en CamelCase

```
class UneBonneClasse;
class uneMauvaiseclasse;
```

2.7 Nom des variables

Toutes les variables sont écrites en **camelCase** [3]

```
T uneBonneVariable;
T UneMauvaiseVariable;
T uneautreMauvaisevariable;
```

et possède des noms le plus explicite possible

```
T nb; // OK
T n; // NOK
T waveLength; // OK
```

Les variables de classes possèdent le même nom que le paramètre de constructeur qui l'initialisera. Le langage nous donne la possibilité de désambiguïser l'utilisation de ces variables à l'aide de

```
T var;
this->var;
```

3. https://fr.wikipedia.org/wiki/CamelCase

et de la liste d'initialisation

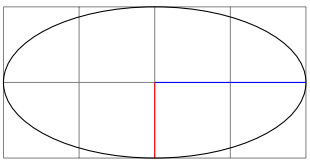
```
UneClasse::UneClasse(T param) : param{param} {}
```

Chapitre 3

Présentation succincte des classes

3.1 geometry

3.1.1 ellipse.hpp



Une ellipse est un objet géométrique à deux dimensions représentée par une courbe plane fermée obtenu par découpe d'un cône sur un plan. Si ce dernier est perpendiculaire à l'axe du cône, l'ellipse sera alors un cercle. Éléments caractéristiques d'une ellipse :

- une coordonnée cartésienne de son centre,
- une distance séparant le centre de l'intersection avec une parallèle à l'axe des ordonnées tangente à l'ellipse voulue,
- une distance séparant le centre de l'intersection avec une parallèle à l'axe des abscisse tangente à l'ellipse voulue.

Ces éléments nous permettront de tracer une ellipse selon cette équation :

$$\frac{(x - c_x)^2}{{x_{radius}}^2} + \frac{(y - c_y)^2}{{y_{radius}}^2} = 1$$

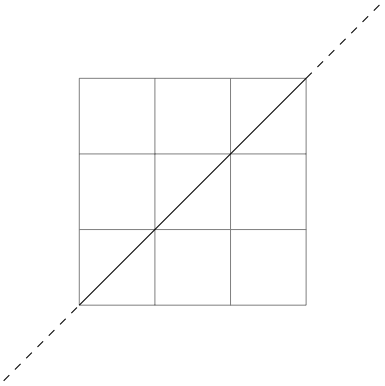
Cette classe peut tout à fait être instancié en objet géométrique elliptique et possède des méthodes d'interactions avec une droite.

```
Ellipse::getIntersectionsPoints(Line);
```

Dans le contexte présent, certains éléments du jeu seront des ellipse par le phénomène d'héritage mis en place dans le paradigme orienté objet de C++ [1]

1. http://www.tutorialspoint.com/cplusplus/cpp_object_oriented.htm

3.1.2 line.hpp



Une droite est une ligne sans épaisseur, rectiligne et infinie dans le plan. Pour exister, une droite aura besoin :

- d'un coefficient angulaire $m = \frac{\Delta y}{\Delta x}$ représentant la distance à parcourir sur l'axe des ordonnées pour une unité de distance sur l'axe des abscisses.
- d'un terme indépendant $p = \frac{y}{m \cdot x}$ représentant le décalage de chaque point sur l'axe des ordonnées,
- ou de deux points de coordonnées dans le plan,
- ou d'un point de coordonnées dans le plan et d'un coefficient angulaire.

Ces éléments nous permettent de tracer une droite selon cette équation :

$$y = m \cdot x + p$$

3.1.3 rectangle.hpp

Un rectangle est une forme géométrique à 4 segments de droite [2] parallèle deux à deux.Ceux-ci vont donc former 4 angles droit ($\frac{\pi}{2} rad$) Cette forme peut être représentée par :

- la coordonnée du coin supérieur gauche $Sg = (x, y)$
- la grandeur des deux segments formant un angle de $\frac{\pi}{2} rad$ en ce point $hauteur$ et $largeur$.

Ainsi, il sera aisé de déterminer la position des autres coins

- $Sd = (Sg_x + largeur, Sg_y)$
- $Ig = (Sg_x, Sg_y + hauteur)$
- $Id = (Sg_x + largeur, Sg_y + hauteur)$

2. Un segment de droite est une partie de droite délimitée par deux points non confondus

Il existe plusieurs cas particuliers de droite :
- une droite parallèle à l'axe des ordonnées qui aura pour équation $x = a$, $a \in \mathbb{R}$,
- une droite parallèle à l'axe des abscisses qui aura pour équation $y = b$, $b \in \mathbb{R}$,

3.1.4 point.hpp

Un point est un objet mathématique permettant de situer un élément dans un plan ou dans l'espace. Dans notre cas, plus spécifiquement dans un plan à deux dimensions. Celui-ci peut-être représenté de plusieurs manières dans le plan cartésien [3] :
- sous la forme d'une coordonnées cartésienne à l'aide de
  - une origine,
  - deux vecteurs partant de cette origine et perpendiculaires,

$$P = (\vec{x}, \vec{y})$$

- et sous la forme d'une coordonnée polaire à l'aide de
  - une origine,
  - une coordonnée radiale r,
  - une coordonnée angulaire α.

3.1.5 utilities.hpp

Le namespace "utilities" mis en place ici est un ensemble de fonctions et valeurs constantes spécifiquement définies pour les calculs intervenant dans le projet.

constantes :

PI est un approximation de π sur 26 décimales,

PI_2 est une approximation de $\frac{\pi}{2}$ sur 26 décimales,

PI_4 est une approximation de $\frac{\pi}{4}$ sur 26 décimales,

EPSILON est une marge d'erreur de 10^{-7},

INF représente un nombre dit "infini" dans le milieu informatique.

fonctions :

Resolution d'équation du second degre

```
utilitaire::secondDegreeEquationSolver
```

3. http://www.cslaval.qc.ca/sitsatlll/maths2003/cartesien.html

Transforme un angle exprime en radian en un angle exprime en degres

```
utilitaire::angleAsDegree
```

Permet de tester l egalite ou l inegalite entre deux nombre reels a un Epsilon d erreur

```
utilitaire::equals
utilitaire::greaterOrEquals
utilitaire::lessOrEquals
```

Permet de trouver le coefficiant angulaire a partir de deux points"

```
utilitaire::slopeFromPoints
```

Permet de trouver la valeur tangante d'un angle en radian mais aussi de retourner une valeur particuliere pour la tangante de $\pi/2$

```
utilitaire::tan
```

Permet de savoir si α vaut $\frac{\pi}{2} + n * \pi \; n \in \mathbb{N}$

```
utilitaire::isHalfPiPlusNPi
```

3.2 éléments

3.2.1 element.hpp

Cette classe, abstraite et donc non instanciable, représente un élément du jeu lié à un, et un seul, niveau du jeu. Cette classe permet donc d'établir une communication entre les différents éléments du niveau et le niveau lui-même à travers un référencement de ce dernier. Par le phénomène d'héritage, chaque élément pourra établir sa manière propre de réagir avec le niveau en lui exprimant :

- ses points d'intersection avec un rayon si il lui sont demandés,

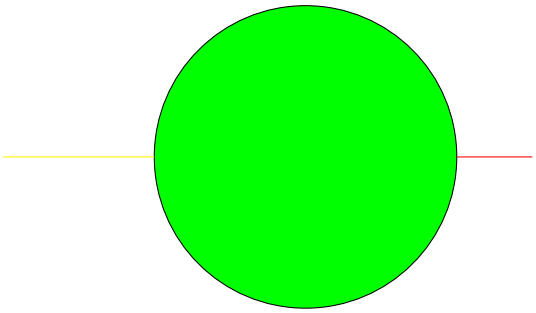
```
Element::includeRay(Ray);
```

- les actions à effectuer si un contact avec le rayon a eu lieu.

```
Element::reactToRay(Ray);
```

Il sera donc aisé pour le niveau de gérer les collisions des éléments de manière anonyme et optimale (cf Niveau).

3.2.2 crystal.hpp

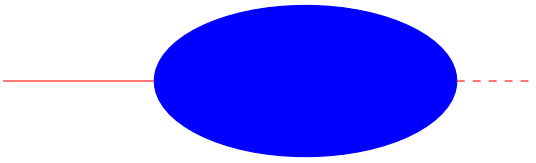


Cette classe est, par héritage, un élément ainsi qu'une ellipse. Il peut donc être modéliser graphiquement, entant que cette dernière, dans le plan et communique avec le niveau auquel il est lié. Le cristal modifie la longueur d'onde d'un rayon qui le traverse. D'un point de vue technique, il en relève donc d'un choix personnel, il ne va pas amplifier le rayon entrant à partir de sa sortie mais arrêtera ce rayon et en créera un nouveau qui, lui, aura subit l'action du cristal.

3.2.3 dest.hpp

Cette classe est, par héritage, un élément ainsi qu'un rectangle. Il peut donc être modéliser graphiquement, entant que cette dernière, dans le plan et communique avec le niveau auquel il est lié. La destination est le but du jeu et l'atteindre terminera automatiquement la partie sur une victoire.

3.2.4 lens.hpp



Cette classe est, par héritage, un élément ainsi qu'un rectangle. Il peut donc être modéliser graphiquement, entant que cette dernière, dans le plan et communique avec le niveau auquel il est lié. La lentille est un éventuel obstacle à un rayon puisqu'il ne le laissera passer que si la longueur d'onde de ce dernier respecte le critère

$$lens_{min} \leq ray_{\alpha} \leq lens_{max}$$

Deux issues sont alors possible :

- la longueur d'onde entre dans l'intervalle et le rayon est renouvelé après la lentille
- ou elle ne rentre pas dans l'intervalle défini et le rayon sera terminé par la lentille

et ce même si le rayon tiré est tangent à la lentille. En effet, si la droite représentant le rayon est tangent à l'ellipse représentant la lentille c'est qu'un point d'intersection existe entre les deux. Il est donc de ce point d'agir comme l'aurait fait l'entière lentille à tout autre point même si cette réaction particulière peut être discutable.

3.2.5 mirror.hpp

Cette classe est, par héritage, un élément ainsi qu'une droite. Le miroir peut donc être modéliser graphiquement, entant que cette dernière, dans le plan et communiquer avec le niveau auquel il est lié. Le miroir est, plus précisément, un segment de droite délimité par deux points. Puisque le miroir effectue une rotation autour d'un point $p \in mirroir$, il n'est pas directement délimité par ses deux extrémités mais bien par sa longueur, son angle, la position absolue (cartésienne) et relative (distance par rapport à une extrémité) de son point de rotation qui définissent, dynamiquement, ses points délimiteurs. Pour trouver les deux points il suffit de :

1. extrémité gauche = (x_{pivot} - position absolue, y_{pivot})
2. extrémité droite = (x_{gauche} + taille du segment, y_{pivot})
3. rotation des deux extrémité autour du point de pivot.

3.2.6 nuke.hpp

Cette classe est, par héritage, un élément ainsi qu'une ellipse. Il peut donc être modéliser graphiquement, entant que cette dernière, dans le plan et communique avec le niveau auquel il est lié. La bombe est une ellipse particulière puisqu'elle possède un $x_{radius} = y_{radius}$ lui donnant comme caractéristique d'être un cercle. Cet objet circulaire est l'objet à éviter lors d'une partie puisqu'il amène directement à la fin d'une partie.

3.2.7 ray.hpp

Cette classe est, par héritage, une droite. Ce rayon représente l'ensemble des points parcouru par un rayon laser et subit les interactions de son environnement comme dans la vie réelle :

– il sera reflété par un miroir selon le principe de réflexion

$$\alpha_i = \alpha_r$$

,

– il sera arrêté par les objets opaques (murs, source, destination).

Le rayon déclenche les réactions des éléments du jeu via la méthode “reactToRay” des héritiers d'Element.

3.2.8 source.hpp

Cette classe est, par héritage, un élément ainsi qu'un rectangle. Il peut donc être modéliser graphiquement, entant que cette dernière, dans le plan et communique avec le niveau auquel il est lié. La source est le point d'émission du tout premier rayon du jeu selon un sens et une direction définis par l'angle d'émission, qui lui est propre, depuis un point donné.

3.2.9 wall.hpp

Cette classe est, par héritage, un élément ainsi qu'une droite. Il peut donc être modéliser graphiquement, entant que cette dernière, dans le plan et communique avec le niveau auquel il est lié. Le mur est défini par deux points qui en font un segment de droite. Il est un objet fixe du jeu qui arrête un rayon et lui défini un point qui le transforme en segment de droite.

3.2.10 level.hpp

3.2.11 levelfactory.hpp

Le créateur de niveau est un "namespace" de méthodes qui va s'occuper de lire les données des éléments dans un fichier. Pour ce faire, il s'occupera de lire chaque type d'objet selon des règles différentes :

```
levelFactory::getSource;
levelFactory::getDestination;
levelFactory::getCrystal;
levelFactory::getLens;
levelFactory::getNuke;
levelFactory::getWall;
levelFactory::getMirror;
```

3.3 exception

3.3.1 starlightexception.hpp

Il est nécessaire, pour bon nombre des classes créées, de valider les arguments passés en paramètre dans le but de ne pas produire d'objets incohérents par rapport à l'analyse préalable du travail à fournir. Pour ce faire, des exceptions doivent être levées quand une instanciation créera un objet non désiré. Cette classe hérite de std : :exception appartenant à la librairie standard. Elle n'a aucune capacité supplémentaire mise à part être spécifique à ce projet.

Les différentes classes pouvant lever cette exception sont :

crystal si la taille de son rayon ne lui permet pas d'exister dans le plan,

lens si son intervalle de longueur d'onde n'est pas cohérent,

level si ses dimensions ne lui permettent pas d'exister dans le plan,

mirror si ses dimensions ne lui permettent pas d'exister dans le plan, si sa position ou son angle n'entre pas dans les limites imposées,

nuke si la taille de son rayon ne lui permet pas d'exister dans le plan,

ray si sa longueur d'onde n'entre pas dans l'intervalle cohérent imposé,

source si sa longueur d'onde n'entre pas dans l'intervalle cohérent imposé,

wall si ses points déterminants ne lui permettent pas d'exister dans le plan,

ellipse si ses dimensions ne lui permettent pas d'exister dans le plan,

rectangle si ses dimensions ne lui permettent pas d'exister dans le plan.

3.4 view

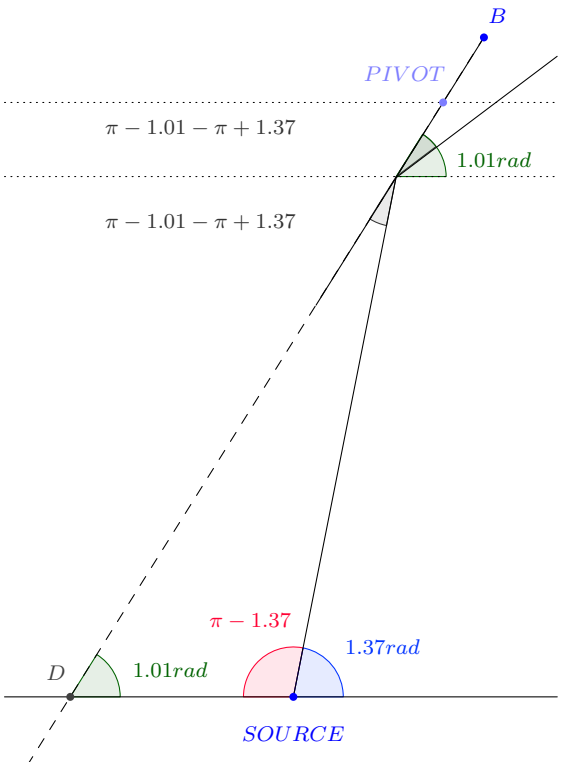
Chapitre 4

Structure générale du programme

Chapitre 5

Détail des algorithmes utilisés

5.1 Algorithme de réflexion



α l'angle du nouveau rayon
β l'angle du miroir
γ l'angle incident
ϵ l'angle de la source

$\gamma = (\pi - \beta - (\pi - \epsilon))$ par la somme des angles intérieurs d'un triangle

valant πrad,
$\alpha = \beta - \gamma$ par les angles correspondants

5.2 Algorithme d'intersection

5.2.1 Intersection de deux droites

5.2.2 Intersection d'une droite et d'un rectangle

5.2.3 Intersection d'une droite et d'une ellipse

Chapitre 6

Test effectués

6.1 Framework de test

Le Framework de test utilisé est "Catch"[1]. Ses avantages sont :
- un simple header est requis,
- **TEST_CASE** créant un bloc de tests,
- **SECTION** créant un sous bloc de tests,
- un ensemble de macros d'assertions sont disponibles :
  - **REQUIRE** qui attend une expression vraie,
  - **REQUIRE_FALSE** qui attend un expression fausse,
  - **REQUIRE_THROWS_AS** qui attend une erreur de type défini,
  - **REQUIRE_NO_THROW** qui n'attend pas d'erreur.

6.2 Tests unitaire

Chaque classe a été soumise à une batterie de tests unitaires. Pour se faire, chacune de ses méthodes s'est vu confirmé son bon fonctionnement au travers de cas dis "limites" et de cas dit "standards".

Les fichiers sources concernés se situe dans dossier "test/" du projet et peuvent être effectués sous certaines condition[2].

1. `https://github.com/philsquared/Catch`
2. en effet, il faudra au préalable rendre la méthode main de l'application inutilisable pour permettre aux tests de s'appliquer exclusivement

Chapitre 7

Conclusion

Annexe A

Références